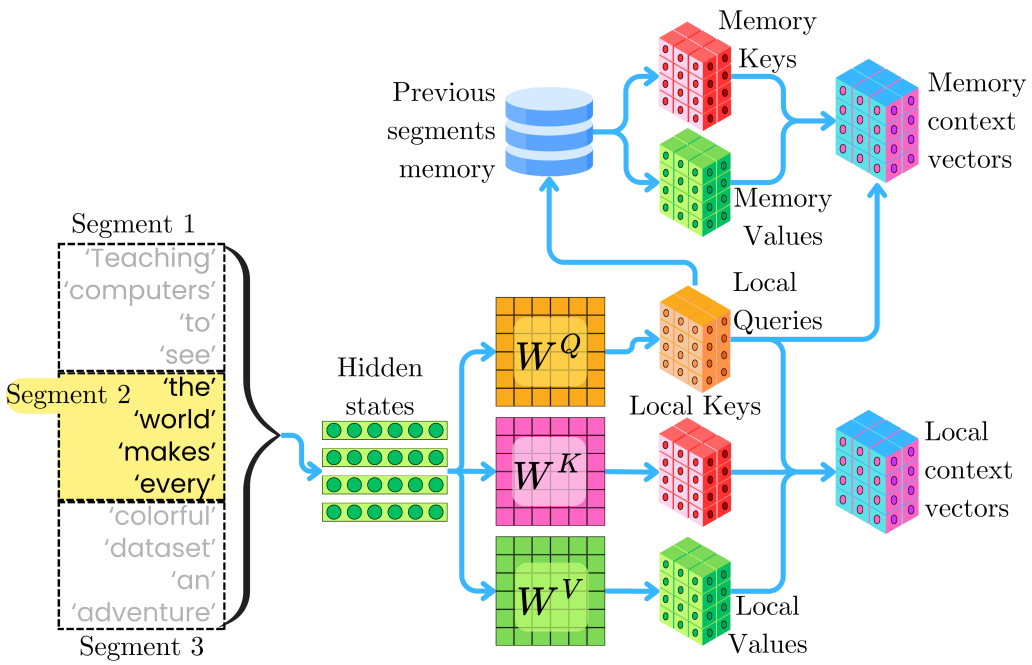
How To Add Memory To The Self-Attention Mechanism

Damien Benveniste
newsletter.TheAiEdge.io



In Transformer-XL, the long-range coherence is captured by the successive layers in the model, but the direct interaction between tokens is lost beyond a two-segment window. Google introduced the *Memorizing Transformers* in 2022 [3] that extended the long-range coherence by caching previous key-value pairs for selective retrieval. The attention computation is broken down into two parts:

- **The local attention:** As before, we partition the input sequence into segments (usually 512 tokens) and compute the token-token interactions within each segment:

$$C_{\text{local}} = \text{Softmax}\left(\frac{Q_{\text{local}}K_{\text{local}}^\top}{\sqrt{d}}\right)V_{\text{local}} \tag{1}$$

  where Q_{local}, K_{local}, and V_{local} are the local queries, keys and values within the segment.

- **The memory attention:** After a segment is processed, the related keys and values are added to a local cache. This cache acts as a memory of the previous segments. Once

a new segment is processed, beside the in-segment attention, we also retrieve key-value pairs $(K_{\text{mem}}, V_{\text{mem}})$ from the memory to provide context from previous segments and capture longer range dependencies:

$$C_{\text{mem}} = \text{Softmax}\left(\frac{Q_{\text{local}} K_{\text{mem}}^{\top}}{\sqrt{d_{\text{head}}}}\right) V_{\text{mem}} \tag{2}$$

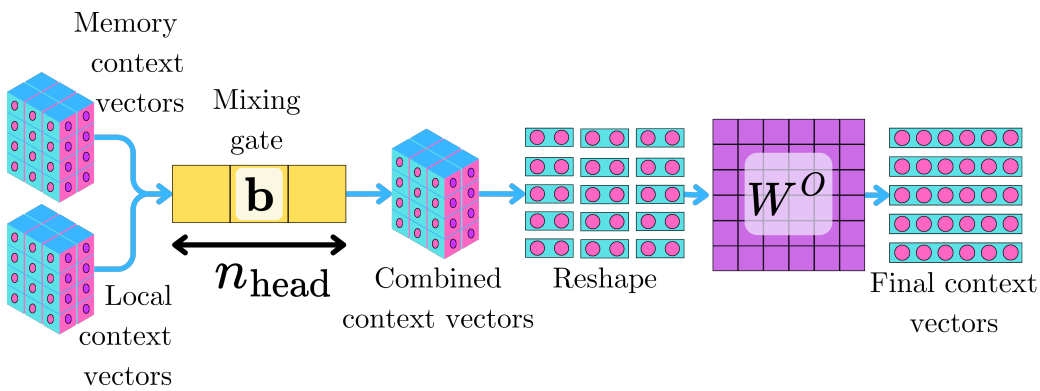
The two resulting sets of context vectors are then simply combined with a weighted average:

$$C = C_{\text{local}} \cdot \sigma(\mathbf{b}) + C_{\text{mem}} \cdot (1 - \sigma(\mathbf{b})) \tag{3}$$

where $\sigma(\mathbf{b})$ is the sigmoid function that acts as a gate for each head, and $\mathbf{b}$ is a learned vector of size n_{head} that will put more or less weight on local vs memory contributions. This gating mechanism allows each head to learn whether to focus more on local context or long-range memory. Once combined, the context vectors are reshaped from $d_{\text{head}} \times n_{\text{head}} \times n$ to $d_{\text{model}} \times n$, where n is the number of tokens per segment, and passed through the last projection matrix W^O as usual:

$$C_{\text{final}} = W^O \cdot \text{Reshape}(C)$$

One important architectural detail is that this is done for only one of attention layers in the model. Applying the memory retrieval mechanism to every layer would significantly increase computation time and resource requirements. Even with just one memory-augmented layer, the paper reports step time increasing from 0.2s to 0.6s when scaling memory from 8K to 65K tokens. In their experiments, they specifically mention using the 9th layer of a 12-layer transformer as the memory-augmented attention layer. Placing the memory layer too early means the representations aren't rich enough to form effective queries, while placing it too late means the retrieved information has less opportunity to influence the final predictions.



Once a segment is processed, the related keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n]$ and values $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$ are stored in the memory:

$$M_{\text{new}} = M_{\text{old}} \cup \{(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_n, \mathbf{v}_n)\} \tag{4}$$

In their experiments, the memory size ranged from 8K to 262K tokens, and if the memory exceeds its maximum size, the oldest entries are removed. When processing a new segment,

for each query $\mathbf{q}_i$, we find the k most similar keys $[\mathbf{k}_1^{\text{mem}}, \ldots, \mathbf{k}_k^{\text{mem}}]$ in memory, where similarity is measured by the dot-product metric:

$$\text{sim}\left(\mathbf{q}_i, \mathbf{k}_j^{\text{mem}}\right) = \mathbf{q}_i^\top \cdot \mathbf{k}_j^{\text{mem}} \tag{5}$$

During training, as the segments are processed, the model parameters are updated and the queries, keys and values representations evolve. To mitigate the effects of distributional shift, the queries and keys are normalized:

$$\hat{\mathbf{q}}_i = \frac{\mathbf{q}_i}{\|\mathbf{q}_i\|}, \quad \hat{\mathbf{k}}_j^{\text{mem}} = \frac{\mathbf{k}_j^{\text{mem}}}{\left\|\mathbf{k}_j^{\text{mem}}\right\|} \tag{6}$$

Normalization constrains all vectors to live on the unit hypersphere, creating a more stable similarity space even as parameters evolve. By normalizing, we emphasize the relative patterns in the vectors rather than their absolute values. Recall that the softmax transformation exacerbates the largest values, therefore the computation of context vectors is dominated by the terms with highest $\mathbf{q}_i^\top \cdot \mathbf{k}_j^{\text{mem}}$ product:

$$\mathbf{c}_i^{\text{mem}} = \sum_{j=1}^{k} \text{Softmax}\left(\mathbf{q}_i^\top \cdot \mathbf{k}_j^{\text{mem}}\right) \mathbf{v}_j^{\text{mem}} \tag{7}$$

motivating the search for the top-k similar keys in the memory. They found that $k = 128$ led to the best results, while $k = 32$ provided less computational requirements with minimal loss in performance. To retrieve the top-k similar keys, they used an approximate k-Nearest Neighbor (kNN) approximate algorithm such as ScaNN [1] and Faiss [2] guaranteeing a retrieval time $\sim \mathcal{O}(\log N)$ for 90% recall performance.

As for the Transformer-XL, to prevent the unbounded growth of the memory during the backward pass, the gradients are not backpropagated into the memory. The backward pass does not require storing computation graphs for the entire memory, only for the k retrieved items per query. The external memory content (keys and values) is treated as non-differentiable constants (we do not update W^K and W^V), and the gradients flow through the operations performed using those retrieved items to update the query computation (to update W^Q).

When processing a new segment, the first token only has access to the external memory and its query vector is formed with minimal context, potentially reducing retrieval quality. This problem can be addressed by combining the Memorizing Transformer with Transformer-XL's caching mechanism where we cache the hidden states of the previous segment to provide better context to the initial tokens. Overall, this led to better coherence than Transformer-XL over very long documents. One of the most significant findings from the Memorizing Transformer research was the dramatic parameter efficiency it achieved. The experiments demonstrated that adding even a modest memory component (8K tokens) to a 200M parameter model yielded better perplexity than a vanilla transformer with 1B parameters, representing a 5× parameter efficiency gain. This efficiency held across model scales, suggesting that the ability to explicitly store and retrieve information provides a fundamentally different advantage than simply encoding knowledge implicitly in more model parameters. This finding challenges the scaling paradigm in language modeling, indicating that architectural innovations focused on memory and retrieval may offer more efficient paths to performance improvement than simply increasing model size.

References

[1] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization, 2020.

[2] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus, 2017.

[3] Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers, 2022.